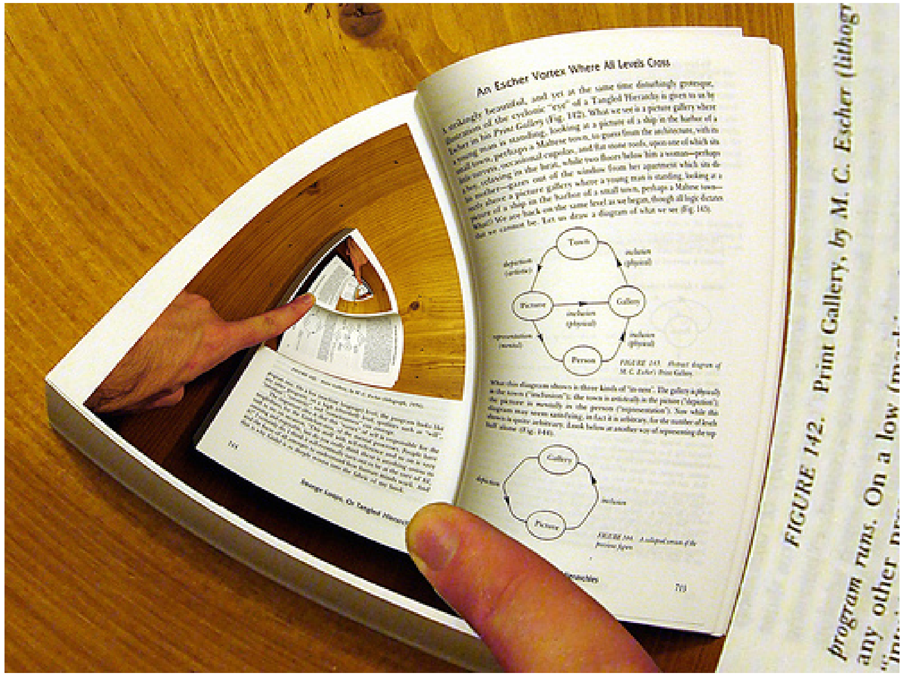
CSE 505: Programming Languages

Lecture 16 —
Evaluation Contexts
First-Class Continuations
Continuation-Passing Style

Zach Tatlock
Winter 2015

GOTO the past / programs choose their own adventure.



But first, some clean up.

Our semantics:

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \qquad \frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \qquad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')}$$

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')} \qquad \frac{e \to e'}{e.1 \to e'.1} \qquad \frac{e \to e'}{e.2 \to e'.2}$$

$$\frac{e \to e'}{\textbf{match } e \textbf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to \textbf{match } e' \textbf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2}$$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{}{(v_1, v_2).1 \to v_1} \qquad \frac{}{(v_1, v_2).2 \to v_2}$$

$$\frac{}{\textbf{match } \mathsf{A}(v) \textbf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to e_1[v/x]}$$

$$\frac{}{\textbf{match } \mathsf{B}(v) \textbf{ with } \mathsf{A}y.\ e_1 \mid \mathsf{B}x.\ e_2 \to e_2[v/x]}$$

But first, some clean up.

Our semantics:

Boring rules to grind sub-expressions down:

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \qquad \frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \qquad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')}$$

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')} \qquad \frac{e \to e'}{e.1 \to e'.1} \qquad \frac{e \to e'}{e.2 \to e'.2}$$

$$\frac{e \to e'}{\textbf{match } e \textbf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to \textbf{match } e' \textbf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2}$$

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{}{(v_1, v_2).1 \to v_1} \qquad \frac{}{(v_1, v_2).2 \to v_2}$$

$$\frac{}{\textbf{match } \mathsf{A}(v) \textbf{ with } \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2 \to e_1[v/x]}$$

$$\frac{}{\textbf{match } \mathsf{B}(v) \textbf{ with } \mathsf{A}y.\ e_1 \mid \mathsf{B}x.\ e_2 \to e_2[v/x]}$$

But first, some clean up.

Our semantics:

Boring rules to grind sub-expressions down:

$$\frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'} \qquad \frac{e \to e'}{\mathsf{A}(e) \to \mathsf{A}(e')} \qquad \frac{e \to e'}{\mathsf{B}(e) \to \mathsf{B}(e')}$$

$$\frac{e_1 \to e_1'}{(e_1, e_2) \to (e_1', e_2)} \qquad \frac{e_2 \to e_2'}{(v_1, e_2) \to (v_1, e_2')} \qquad \frac{e \to e'}{e.1 \to e'.1} \qquad \frac{e \to e'}{e.2 \to e'.2}$$

$$\frac{e \to e'}{\textbf{match } e \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \to \textbf{match } e' \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2}$$

Interesting rules that actually do work:

$$\frac{}{(\lambda x.\ e)\ v \to e[v/x]} \qquad \frac{}{(v_1, v_2).1 \to v_1} \qquad \frac{}{(v_1, v_2).2 \to v_2}$$

$$\frac{}{\textbf{match A}(v) \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \to e_1[v/x]}$$

$$\frac{}{\textbf{match B}(v) \textbf{ with A}y.\ e_1 \mid \textbf{B}x.\ e_2 \to e_2[v/x]}$$

We can do better: Separate concerns

Evaluation contexts define where interesting work can happen:

$$\begin{array}{rcl} E & ::= & [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \\ & \mid & \mathsf{A}(E) \mid \mathsf{B}(E) \mid (\textbf{match } E \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2) \end{array}$$

How many $[\cdot]$ ("holes") can an evaluation context have? Only one.

$E[e]$ just means to "fill the hole" in E with e:

$([\cdot].1)[(1,2)] = (1,2).1$

$([\cdot],\ \lambda x.x)[1] = (1,\ \lambda x.x)$

$([\cdot]\ x\ y)[\lambda a.\ \lambda b.\ b\ a] = (\lambda a.\ \lambda b.\ b\ a)\ x\ y$

We can do better: Separate concerns

Evaluation contexts define where interesting work can happen:

$$\begin{array}{rcl} E & ::= & [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \\ & \mid & \mathsf{A}(E) \mid \mathsf{B}(E) \mid (\textbf{match } E \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2) \end{array}$$

$E[e]$ just means to "fill the hole" in E with e.

Now we can cleanly separate our semantics:

$e \to e'$ with 1 rule: $\dfrac{e \xrightarrow{\text{p}} e'}{E[e] \to E[e']}$

$e \xrightarrow{\text{p}} e'$ does all the "interesting work":

$$\frac{}{(\lambda x.\ e)\ v \xrightarrow{\text{p}} e[v/x]} \qquad \frac{}{(v_1, v_2).1 \xrightarrow{\text{p}} v_1} \qquad \frac{}{(v_1, v_2).2 \xrightarrow{\text{p}} v_2}$$

$$\frac{}{\textbf{match A}(v) \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2 \xrightarrow{\text{p}} e_1[v/x]}$$

$$\frac{}{\textbf{match B}(v) \textbf{ with A}y.\ e_1 \mid \textbf{B}x.\ e_2 \xrightarrow{\text{p}} e_2[v/x]}$$

Evaluation with evaluation contexts

$$\begin{array}{rcl} E & ::= & [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \\ & \mid & \mathsf{A}(E) \mid \mathsf{B}(E) \mid (\textbf{match } E \textbf{ with A}x.\ e_1 \mid \textbf{B}y.\ e_2) \end{array}$$

Evaluation relies on *decomposition* (unstapling the correct subtree)

- Given e, find E, e_a, e_a' such that $e = E[e_a]$ and $e_a \xrightarrow{\text{p}} e_a'$

Many possible eval contexts may match a given e ...

$$\begin{array}{rcl} ([\cdot])[(1,(1,(1,(1,1))))] & = & (1,(1,(1,(1,1)))) \\ ((1,[\cdot]))[(1,(1,(1,1)))] & = & (1,(1,(1,(1,1)))) \\ ((1,(1,[\cdot])))[(1,(1,1))] & = & (1,(1,(1,(1,1)))) \\ ((1,(1,(1,[\cdot]))))[(1,1)] & = & (1,(1,(1,(1,1)))) \\ ((1,(1,(1,(1,[\cdot])))))[1] & = & (1,(1,(1,(1,1)))) \end{array}$$

Evaluation with evaluation contexts

$$\begin{array}{rcl} E & ::= & [\cdot] \mid E\ e \mid v\ E \mid (E, e) \mid (v, E) \mid E.1 \mid E.2 \\ & \mid & \mathsf{A}(E) \mid \mathsf{B}(E) \mid (\mathbf{match}\ E\ \mathbf{with}\ \mathsf{A}x.\ e_1 \mid \mathsf{B}y.\ e_2) \end{array}$$

Evaluation relies on *decomposition* (unstapling the correct subtree)

- Given e, find E, e_a, e'_a such that $e = E[e_a]$ and $e_a \xrightarrow{\mathrm{p}} e'_a$

Unique Decomposition Theorem: at most one decomposition of e

- E carefully picks leftmost non-value sub-expression
- Hence eval is deterministic: at most one primitive step applies

Progress Theorem (restated): If e is well-typed, then there is a decomposition or e is a value

Evaluation Contexts: So what?

Small-step semantics (old) and evaluation-context semantics (new) are *very* similar:

- Totally equivalent step sequence
  - (made both left-to-right call-by-value)
- Just rearranged things to be more concise: Each boring rule became a form of E
- Both "work" the same way:
  - Find the next place in the program to take a "primitive step"
  - Take that step
  - Plug the result into the rest of the program
  - Repeat (next "primitive step" could be somewhere else) until you can't anymore (value or stuck)

Evaluation contexts so far just cleanly separate the "find and plug" from the "take that step" by building an explicit E

Continuations

Now that we have defined E explicitly in our *metalanguage*, what if we also put it on our *language*

- From metalanguage to language is called *reification*

First-class continuations:

$$\begin{array}{rcl} e & ::= & \ldots \mid \mathbf{letcc}\ x.\ e \mid \mathbf{throw}\ e\ e \mid \mathbf{cont}\ E \\ v & ::= & \ldots \mid \mathbf{cont}\ E \\ E & ::= & \ldots \mid \mathbf{throw}\ E\ e \mid \mathbf{throw}\ v\ E \end{array}$$

$$\frac{}{E[\mathbf{letcc}\ x.\ e] \to E[(\lambda x.\ e)(\mathbf{cont}\ E)]} \qquad \frac{}{E[\mathbf{throw}\ (\mathbf{cont}\ E')\ v] \to E'[v]}$$

- New operational rules for $\to$ not $\xrightarrow{\mathrm{p}}$ because "the E matters"
- **letcc** $x.\ e$ grabs the current evaluation context ("the stack")
- **throw** (**cont** E') v restores old context: "jump somewhere"
- **cont** E not in source programs: "saved stack (value)"

Examples (exceptions-like)

$$1 + (\mathbf{letcc}\ k.\ 2 + 3) \to^* 6$$

$$1 + (\mathbf{letcc}\ k.\ 2 + (\mathbf{throw}\ k\ 3)) \to^* 4$$

$$1 + (\mathbf{letcc}\ k.\ (\mathbf{throw}\ k\ (2 + 3))) \to^* 6$$

$$1 + (\mathbf{letcc}\ k.\ (\mathbf{throw}\ k\ (\mathbf{throw}\ k\ (\mathbf{throw}\ k\ 2)))) \to^* 3$$

Another view

If you're confused, think call stacks:

- What if your favorite language had operations for:
  - Store current stack in `x`
  - Replace current stack with stack in `x`
- "Resume the stack's hole" with something different or when mutable state is different
  - Else you are sure to have an infinite loop since you will later resume the stack again

Example ("time travel")

SML/NJ has continuations. This runs and binds 10 to `z`:

```
open SMLofNJ.Cont
val g : int cont option ref = ref NONE
val x = ref true (* avoids infinite loop *)
val y = ref (1 + 2 + (callcc (fn k => ((g := SOME k); 3))))
val z = if !x then (x := false; throw (valOf (!g)) 7) else !y
```


Is this useful?

First-class continuations are a *single* construct sufficient for:

- Exceptions
- Cooperative threads (including coroutines)
  - "yield" captures the continuation (the "how to resume me") and gives it to the scheduler (implemented in the language), which then throws to another thread's "how to resume me"
- Other crazy things
  - Often called the "goto of functional programming" — incredibly powerful, but nonstandard uses are usually inscrutable
  - Key point is that we can "jump back in" unlike boring-old exceptions

Where are we

Done:

- Redefined our operational semantics using evaluation contexts
- That made it easy to define first-class continuations
- Example uses of continuations

Now: How the heck do we implement this?

Rather than adding a powerful primitive, we can achieve the same effect via a *whole-program translation* into a sublanguage (source-to-source transformation)

- Every function takes extra arg: *continuation* says what's next
- Never "return" — instead call current continuation w/ result
- *Every expression becomes a continuation-accepting function*
- Will be able to reintroduce **letcc** and **throw** "for free"

CPS examples

Invariant: *every* function takes continuation as extra argument

CPS examples

Invariant: *every* function takes continuation as extra argument

```
let mult' ...
```


CPS examples

Invariant: *every* function takes continuation as extra argument

```
let mult' x y k = ...
```


CPS examples

Invariant: *every* function takes continuation as extra argument

```
let mult' x y k = k (x * y)
```

CPS examples

Invariant: *every* function takes continuation as extra argument

```
let mult' x y k = k (x * y)
let add'  x y k = k (x + y)
let sub'  x y k = k (x - y)
let eq'   x y k = k (x = y)
```


CPS examples

Invariant: *every* function takes continuation as extra argument

```
let mult' x y k = k (x * y)
let add'  x y k = k (x + y)
let sub'  x y k = k (x - y)
let eq'   x y k = k (x = y)

let rec fact' n k = ...
```


CPS examples

Invariant: *every* function takes continuation as extra argument

```
let mult' x y k = k (x * y)
let add'  x y k = k (x + y)
let sub'  x y k = k (x - y)
let eq'   x y k = k (x = y)

let rec fact' n k =
  (eq' n 0 (fun b ->
    (if b then
      (k 1)
     else
      (sub' n 1 (fun m ->
        (fact' m (fun p ->
          (mult' n p k)))))))))
```


CPS examples

OK, now you convert :

```
let fact n =
  aux n 1

let rec aux n acc =
  if n = 0 then
    acc
  else
    aux (n - 1) (n * acc)
```

The CPS transformation (one way to do it)

A metafunction from expressions to expressions

Example source language (other features similar):

$$\begin{array}{rcl} e & ::= & x \mid \lambda x.\ e \mid e\ e \mid c \mid e + e \\ v & ::= & x \mid \lambda x.\ e \mid c \end{array}$$

$$\begin{array}{rcl} \mathbf{CPS_E}(v) & = & \lambda k.\ k\ \mathbf{CPS_V}(v) \\ \mathbf{CPS_E}(e_1 + e_2) & = & \lambda k.\ \mathbf{CPS_E}(e_1)\ \lambda x_1.\ \mathbf{CPS_E}(e_2)\ \lambda x_2.\ k\ (x_1{+}x_2) \\ \mathbf{CPS_E}(e_1\ e_2) & = & \lambda k.\ \mathbf{CPS_E}(e_1)\ \lambda f.\ \mathbf{CPS_E}(e_2)\ \lambda x.\ f\ x\ k \end{array}$$

$$\begin{array}{rcl} \mathbf{CPS_V}(c) & = & c \\ \mathbf{CPS_V}(x) & = & x \\ \mathbf{CPS_V}(\lambda x.\ e) & = & \lambda x.\ \lambda k.\ \mathbf{CPS_E}(e)\ k \end{array}$$

To run the whole program e, do $\mathbf{CPS_E}(e)\ (\lambda x.\ x)$

Result of the CPS transformation

- Correctness: e is equivalent to $\mathbf{CPS_E}(e)\ \lambda x.\ x$
- If whole program has type τ_P and e has type τ, then $\mathbf{CPS_E}(e)$ has type $(\tau \to \tau_P) \to \tau_P$
- Fixes evaluation order: $\mathbf{CPS_E}(e)$ will evaluate e in left-to-right call-by-value
  - Other similar transformations encode other evaluation orders
  - Every intermediate computation is bound to a variable (helpful for compiler writers)
- For all e, evaluation of $\mathbf{CPS_E}(e)$ stays in this sublanguage:

$$\begin{array}{rcl} e & ::= & v \mid v\ v \mid v\ v\ v \mid v\ (v + v) \\ v & ::= & x \mid \lambda x.\ e \mid c \end{array}$$

- Hence no need for a call-stack: every call is a tail-call
  - Now the *program* is maintaining the evaluation context via a closure that has the next "link" in its environment that has the next "link" in *its* environment, etc.

Encoding first-class continuations

If you apply the CPS transform, then you can add **letcc** and **throw** "for free" right in the source language

$$\begin{array}{rcl} \mathbf{CPS_E}(\textbf{letcc}\ k.\ e) & = & \lambda k.\ \mathbf{CPS_E}(e)\ k \\ \mathbf{CPS_E}(\textbf{throw}\ e_1\ e_2) & = & \lambda k.\ \mathbf{CPS_E}(e_1)\ \lambda x_1.\ \mathbf{CPS_E}(e_2)\ \underbrace{\lambda x_2.\ x_1\ x_2}_{\text{or just } x_1} \end{array}$$

- **letcc** gets passed the current continuation just as it needs
- **throw** ignores the current continuation just as it should

You can also manually program in this style (fully or partially)

- Has other uses as a programming idiom too...

A useful advanced programming idiom

- A first-class continuation can "reify session state" in a client-server interaction
  - If the continuation is passed to the client, which returns it later, then the server can be stateless
  - Suggests CPS for web programming
  - Better: tools that do the CPS transformation for you
    - Gives you a "prompt-client" primitive without server-side state
- Because CPS uses only tail calls, it avoids deep call stacks when traversing recursive data structures
  - See `lec13code.ml` for this and related idioms

In short, "thinking in terms of CPS" is a powerful technique few programmers have